WEST Search History

DATE: Friday, November 14, 2003

Set Name side by side	**Query**	**Hit Count**	**Set Name** result set
DB=USPT; PLUR=YES; OP=ADJ			
L16	L15	8	L16
DB=USPT,PGPB; PLUR=YES; OP=ADJ			
L15	L14 and (zip$ or url)	18	L15
L14	L13 and l5	46	L14
L13	L12 and instantia$	167	L13
L12	L11 and search$	308	L12
L11	L7	601	L11
DB=JPAB,EPAB,TDBD; PLUR=YES; OP=ADJ			
L10	L8	7	L10
DB=JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ			
L9	L8 and search$	3	L9
L8	L7	20	L8
DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ			
L7	L6 and (stream$ or bytecode or byte code or byte?code or Java?code or Javacode or Java code)	621	L7
L6	classload$ or class?load$ or class load$	1012	L6
DB=USPT,PGPB; PLUR=YES; OP=ADJ			
L5	L4 or l3 or l2 or l1	9948	L5
L4	((713/2)!.CCLS.)	876	L4
L3	((707/2 |707/3 |707/4 |707/5 |707/6)!.CCLS.)	5823	L3
L2	((717/107 |717/108 |717/109 |717/147 |717/148 |717/162 |717/163 |717/164 |717/165 |717/166 |717/167 |717/168)!.CCLS.)	1192	L2
L1	((709/231 |709/311 |709/312 |709/313 |709/314 |709/315 |709/331 |709/332)!.CCLS.)	2274	L1

END OF SEARCH HISTORY

WEST

Generate Collection | Print

Search Results - Record(s) 1 through 7 of 7 returned.

☐ 1. Document ID: JP 2000132388 A

L10: Entry 1 of 7 File: JPAB May 12, 2000

PUB-NO: JP02000132388A
DOCUMENT-IDENTIFIER: JP 2000132388 A
TITLE: METHOD AND DEVICE FOR PROCESSING AND DISTRIBUTING SOFTWARE COMPONENT

PUBN-DATE: May 12, 2000

INVENTOR-INFORMATION:

NAME	COUNTRY
WEBB, ALAN MICHAEL	

INT-CL (IPC): G06 F 9/06; G06 F 9/445; G09 C 1/00

ABSTRACT:

PROBLEM TO BE SOLVED: To make an idea in software to be more safe by inspecting a software component and decoding a part obtained by detecting a compiled part and ciphering the part when it exists.

SOLUTION: A Java compiler takes out a source code and converts it into a byte code (S1). A post Java compiling processing is applied to a class file and a 'compiled method' attribute is set (S2A). A post Java ciphering processing is applied to the class file and several methods are ciphered (S2B). The changed class file is taken out from a server or a storage device and it is cache-stored (S3). A JVM(Java virtual machine) class loader verifies the byte code by inspecting a syntax agasint a Java language specification and prepares the class object from the changed class file and solves a symbol in the class object (S4, 5 and 6).

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

☐ 2. Document ID: EP 1174796 A2

L10: Entry 2 of 7 File: EPAB Jan 23, 2002

PUB-NO: EP001174796A2
DOCUMENT-IDENTIFIER: EP 1174796 A2
TITLE: System and method for instrumenting application class files

PUBN-DATE: January 23, 2002

INVENTOR-INFORMATION:

NAME	COUNTRY
BERRY, ROBERT FRANCIS	GB
GU, WEIMING	GB
HUSSAIN, RIAZ Y	GB
LEVINE, FRANK ELIOT	GB
WONG, WAI YEE PETER	GB

INT-CL (IPC): G06 F 11/34
EUR-CL (EPC): G06F011/34; G06F011/00, G06F011/00

ABSTRACT:

CHG DATE=20020202 STATUS=O> The present invention is directed to a system and method for modifying a class file for the purpose of instrumenting without requiring separate files to correlate the instrumentation. A class file is instrumented with hooks. Each hook is injected in a method at a critical point in the code for tracking path flow, such as where the method will be entered or exited. Each hook includes an identifier to identify for the method in which it is injected. Rather than using the method's name, hooks use unique major and minor codes to identify the method. Static initializers are declared for the class to output other hooks identifying the methods being instrumented. When a class is loaded, the static initializers are executed and hooks identifying the method name and the major and minor codes for each instrumented method are output to, for instance, a trace record. Then, when a method is entered or exited, the hooks identifying the entry or exit are also outputted to a trace record. When postprocessing the trace records, class and instrumentation correlation information is available for merging from trace records in the trace stream.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

---

3. Document ID: EP 778520 A2

L10: Entry 3 of 7 File: EPAB Jun 11, 1997

PUB-NO: EP000778520A2
DOCUMENT-IDENTIFIER: EP 778520 A2
TITLE: System and method for executing verifiable programs with facility for using non-verifiable programs from trusted sources

PUBN-DATE: June 11, 1997

INVENTOR-INFORMATION:

NAME	COUNTRY
MCMANIS, CHARLES E	US

INT-CL (IPC): G06 F 9/45; G06 F 9/46; G06 F 1/00
EUR-CL (EPC): G06F009/44; G06F001/00, G06F001/00 , G06F009/445

ABSTRACT:

CHG DATE=19990617 STATUS=O> A computer system includes a program executer that executes verifiable architecture neutral programs and a class loader that prohibits the loading and execution of non-verifiable programs unless (A) the non-verifiable program resides in a trusted repository of such programs, or (B) the non-verifiable program is indirectly verifiable by way of a digital signature on the non-verifiable program that proves the program was produced by a trusted source. In the preferred embodiment, verifiable architecture neutral programs are Java bytecode programs whose integrity is verified using a Java bytecode program verifier. The non-verifiable programs are generally architecture specific compiled programs generated with the assistance of a compiler. Each architecture specific program typically includes two signatures, including one by the compiling party and one by the compiler. Each digital signature includes a signing party identifier and an encrypted message. The encrypted message includes a message generated by a predefined procedure, and is encrypted using a private encryption key associated with the signing party. A digital signature verifier used by the class loader includes logic for processing each digital signature by obtaining a public key associated with the signing party, decrypting the encrypted message of the digital signature with that public key so as generate a decrypted message, generating a test

message by executing the predefined procedure on the architecture specific program associated with the digital signature, comparing the test message with the decrypted message, and issuing a failure signal if the decrypted message digest and test message digest do not match.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 4. Document ID: NNRD454160

L10: Entry 4 of 7 File: TDBD Feb 1, 2002

TDB-ACC-NO: NNRD454160

DISCLOSURE TITLE: Moving data from an HTTP request object to a data object with name retention in Java.

PUBLICATION-DATA:
IBM technical Disclosure Bulletin, February 2002, UK

ISSUE NUMBER: 454
PAGE NUMBER: 326

DISCLOSURE TEXT:

This disclosure describes an algorithm for moving data from an HTTP Request object to a data object with name retention in Java. Based on a simple naming scheme, it is possible to automate the mapping of HTTP Request data to attributes in a Java data object, thus removing the need for programmer-level mapping of each individual value. This generic algorithm can support multiple data objects as well as data objects with various numbers of attributes, provided the naming scheme is adhered to. In a typical web application, a user enters data into a web page consisting of an input form. When data entry is complete, the user then submits the form to a server over HTTP. The information provided to the server by a user's browser is made available through the Request object. It is then the server's responsibility to retrieve the applicable data from the Request object. Most of the time, data sent by a user is gathered from the Request object and retained in a data object for subsequent processing and/or persistent storage. To transfer data from the Request object to a data object, server-side applications must know the name of the parameter to retrieve from the Request object and the specific data object field it corresponds to. For each piece of information sent by the user that needs to be stored in the data object, explicit mapping of the Request parameter to a data object field is needed. For web applications that utilize many data objects in such a manner, or for data objects that contain many fields, this requirement of explicitly mapping Request parameters to data object fields is time-consuming and cumbersome. Instead of explicit data mapping, a naming scheme is adopted to identify the mapping of an input form field to a data object attribute. This naming scheme is a prerequisite to the data movement algorithm. A Java class representing a data object must make its data accessible via accessor methods (getter/setter methods). Accessor methods prefix the attribute name with the words "get" and "set" for getter methods and setter methods respectively. Input elements of an HTML form (i.e. text and selection boxes) must be named precisely as the data object attribute it corresponds to. In other words, an attribute must have the same name in the input form definition as in the data object definition (see Figure 1.1). The method for moving data from an HTTP Request object to a data object with name retention exploits the reflection capabilities of the Java language. Reflection enables Java code to dynamically discover information about classes loaded in the current Java Virtual Machine, such as what methods are supported, and to use reflected information to operate on the objects. Once a data object and its matching input form have been defined, the following algorithm can be exercised: Use Java Reflection to retrieve all public member methods implemented by the data object. For each public member method If method name starts with the prefix "set" then If method takes a single parameter then Use Java Reflection to identify the data type of the parameter expected by the method. Compute the parameter name as the method name minus the prefix "set", which is common to all mutator methods. For example, if the

method name is "setFirstName", then the computed parameter name will be "FirstName".

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 5. Document ID: NNRD42092

L10: Entry 5 of 7 File: TDBD Apr 1, 1999

TDB-ACC-NO: NNRD42092

DISCLOSURE TITLE: Debugging JITted Java Code

PUBLICATION-DATA:
Research Disclosure, April 1999, UK

VOLUME NUMBER: 42
ISSUE NUMBER: 420
DISCLOSURE TEXT:

In normal operation in Java, the Java compiler processes Java instructions and generates a .class file which contains byte codes that are interpreted by the Java Virtual Machine when executing any of the methods in that class. But executing interpreted code is considerably slower than executing compiled, or native, code. To solve this problem, the current state of the art in Java Virtual Machine (JVM) technology provides a JIT (Just-In-Time) compiler that translates Java byte codes into native, platform-specific, machine (assembly) code that is executed directly by the client CPU, not interpreted by the JVM. To get to JITted code, two steps must take place. First, the Java instructions (source code in a .java file) need to be compiled by the Java compiler into Java byte codes (and stored in a .class file). The byte codes are loaded when the .class file is loaded by the Java Virtual Machine's class loader. Then when a determination is made that a specific method is to be JITted, the JIT uses the byte codes of the method to create native code (jitcode) to be run when that method is called. Here is an illustration of the process: Java Compiler JIT | | JavaInstruction ------> bytecode ------>jitcode jitcode bytecode ------>jitcode jitcode jitcode jitcode bytecode ------>jitcode jitcode jitcode

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 6. Document ID: NNRD41266

L10: Entry 6 of 7 File: TDBD Aug 1, 1998

TDB-ACC-NO: NNRD41266

DISCLOSURE TITLE: Improving Java's Instanceof Operator: Downloading Classes On Demand

PUBLICATION-DATA:
Research Disclosure, August 1998, UK

VOLUME NUMBER: 41
ISSUE NUMBER: 412
DISCLOSURE TEXT:

This disclosure describes an improvement to the Java* instanceof operator which eliminates the premature downloading of classes that are used as arguments to this operator. The result is a performance improvement. The initial download delay of an applet is decreased since classes are not downloaded until they are actually needed. The following two sub-sections contain background information pertinent to understanding both the problem and the improvement described in this disclosure. A familiarity with Java is assumed. A common deployment model for Java programs involves a client downloading Java applets from a server. The client downloads one or more class files which contain information about each class and the byte-codes which are executed in the client's Java Run-time Environment (JRE). The size and number of class files has a direct impact on performance. Downloading more data from the server means an increased delay when launching a Java applet. One technique for improving this delay is to design a Java applet so that individual functions are downloaded on demand. A large portion of a "download on demand" design will be accomplished as a side effect of following good object-oriented programming techniques. For example, an applet that implements functions X, Y and Z might define several classes. Some classes will only be used for function X, some only for function Y and some only for function Z. Still other classes will be used for all three functions. When the applet is downloaded by the JRE's class loader only the classes that are immediately needed will be downloaded. A class will not be downloaded until the execution of the applet reaches an instruction which requires that class to be downloaded (a "load point"). For example, if function X is invoked by clicking on a button in the GUI then the classes required for function X will be downloaded when the button is clicked. The goal of a "download on demand" design is to delay the downloading of all function-unique classes until the function is invoked for the first time. To achieve this goal the following steps must be taken:
o Identify the functions that will be downloaded on demand.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | KWIC | Draw Desc | Image

---

☐ 7. Document ID: NN9611107

L10: Entry 7 of 7 File: TDBD Nov 1, 1996

TDB-ACC-NO: NN9611107

DISCLOSURE TITLE: Java Dynamic Class Loader

PUBLICATION-DATA:
IBM Technical Disclosure Bulletin, November 1996, US

VOLUME NUMBER: 39
ISSUE NUMBER: 11
PAGE NUMBER: 107 - 108

DISCLOSURE TEXT:

Java Dynamic Class Loader Java is a portable, interpreted, high-performance, simple, object-oriented programming language environment developed at Sun Microsystems. The next generation of network centric applications needs to download code from either the network or the local disk. Java offers facilities for transmitting applications, called applets, over the network and running them on multiple client machines (1,2). Java standard class loader is responsible for loading classes from either the network or the local disk. Once a program instantiates a new class, the class loader fetches the class from the local cache, in case the class has been loaded already, or loads it explicitly. This mechanism is quite efficient because it avoids reloading the same class multiple times. The main drawback is related to the inability of the class loader to detect whether a class has been modified and then load the new version of it. This prevents updating applications written in Java while running. Java does not allow substitution of the standard class loader with a custom class loader, but it allows loading selected classes using a custom class loader. This means that the application is loaded using the standard class loader whereas the user can load selected classes using a custom class loader. Writing a custom class loader that has the ability to detect whether a class is changed and then loading the new class version does not solve the problem. The Java byte code verifier is a component of the Java runtime that checks the byte code, i.e., the class being loaded, and decides whether the class can be instantiated. If a class is to be reloaded, the byte code verifier detects that the implementation of the class, or of some parent classes, has changed and then it refuses to instantiate a new class instance using the new version of the class. The fact that the current class implementation does not match the previous class implementation is interpreted as a security violation. The Dynamic Class Loader (DCL) herein described allows dynamic creation of object instances of classes whose implementation changes during program execution. Being the standard class loader used by default by Java to load the application, the DCL can be used to load additional classes which are not loaded by the standard class loader. In order to do this, each class loaded by the DCL must be derived by a common superclass or interface. The application knows only this superclass and, therefore, it uses the standard class loader to load all the instances. The subclasses will then be loaded by the DCL: this mechanism works since the superclass, which is the only object class/interface known to the application, will not change during the program execution whereas subclasses can change since they have been loaded by the DCL; hence, they are out of control of the Java class loader. In other words, this solution allows the implementation of a class to be changed while its interface is preserved. For instance, suppose having a drawing application, the application knows only about the class DrawingTool and it uses the DCL to load subclasses of the class DrawingTool such as CircleTool and OvalTool. The standard class loader caches internally the classes it knows up to the DrawingTool class in the class inheritance hierarchy. This means that the byte code verifier will accept new class definitions if and only if the classes loaded by the standard class loader are not modified. Hence, the DCL loads classes derived from the DrawingTool class and uses the standard class loader to resolve the classes above it (i.e., load the superclasses). The DCL allows efficiently loading classes and detects when a new class has been changed. Because the main program knows only about the class DrawingTool, the DCL is responsible for loading the classes CircleTool and OvalTool whereas the classes up to DrawingTool (for instance java.lang.System, java.io.PrintStream and DrawingTool itself) are loaded using the standard class loader using the method findSystemClass of the class ClassLoader. References (1) J. Gosling, H. McGilton, The Java Language Environment, A White Paper Sun Microsystems, Inc. (May 1995). (2) Sun Microsystems, Inc., The Java Virtual Machine Specification Release 1.0 Beta (August 1995).

Generate Collection | Print

Terms	Documents
L8	7

Display Format: - Change Format

Previous Page Next Page